# Improving Sampling-based Uncertainty Quantification Performance Through Embedded Ensemble Propagation

**Eric Phipps (etphipp@sandia.gov),**
**Marta D'Elia, Mohamed Ebeida**
**Sandia National Laboratories**
**and**
**Ahmad Rushdi, UC Davis**

**QUIET 2017**
**July 18-21, 2017**

**SAND2017-7005 C**

U.S. DEPARTMENT OF ENERGY | Office of Science

Advanced Simulation & Computing

Equinox

ECP EXASCALE COMPUTING PROJECT

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia LLC, a wholly owned subsidiary of Honeywell International Inc. for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

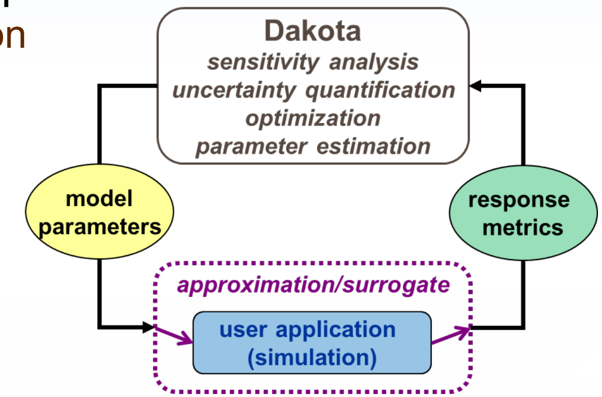
Sandia National Laboratories

# Can Exascale Solve the UQ Challenge?

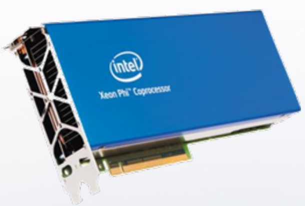
- Focusing on forward propagation of uncertainty through sampling-based methods
  - Common task in many UQ analyses
- Key challenge:
  - Accurately quantifying rare events and localized behavior in high-dimensional uncertain input spaces for expensive simulations
- Can exascale solve this challenge?

# Emerging Architectures Motivate New UQ Approaches

- UQ approaches traditionally implemented as an outer loop:
  - Repeatedly call forward simulation for each sample realization
  - Coarse-grained parallelism over samples
- Increasing UQ performance will require
  - Speeding-up each sample evaluation, and/or
  - Evaluating more samples in parallel
- Many important scientific simulations will struggle with upcoming architectures
  - Irregular memory access patterns
  - Difficulty in exploiting fine-grained parallelism (vectorization, fine-grained threads)
- Increasing UQ parallelism requires exploiting massive increase in on-node parallelism
- Improve performance by propagating multiple samples together at lowest levels of simulation (embedded ensemble propagation)
  - Improve memory access patterns
  - Expose new dimensions of structured fine-grained parallelism
  - Reduce aggregate communication




http://dakota.sandia.gov

# Sparse CRS-Format Matrix-Vector Product

```
// CRS Matrix for an arbitrary floating-point type T
template <typename T>
struct CrsMatrix {
    int num_rows;     // number of rows in matrix
    int num_entries;  // number of nonzeros in matrix
    int *row_map;     // starting index of each row, [0,num_rows+1)
    int *col_entry;   // column indices for each nonzero, [0,num_entries)
    T   *values;      // matrix values of type T, [0,num_entries)
};

// Serial CRS matrix-vector product for arbitrary floating-point type T
template <typename T>
void crs_mat_vec(const CrsMatrix<T>& A, const T *x, T *y) {
    for (int row=0; row<A.num_rows; ++row) {
        const int entry_begin = A.row_map[row];
        const int entry_end   = A.row_map[row+1];
        T sum = 0.0;
        for (int entry = entry_begin; entry < entry_end; ++entry) {
            const int col = A.col_entry[entry];
            sum += A.values[entry] * x[col];
        }
        y[row] = sum;
    }
}
```

# Simultaneous ensemble propagation

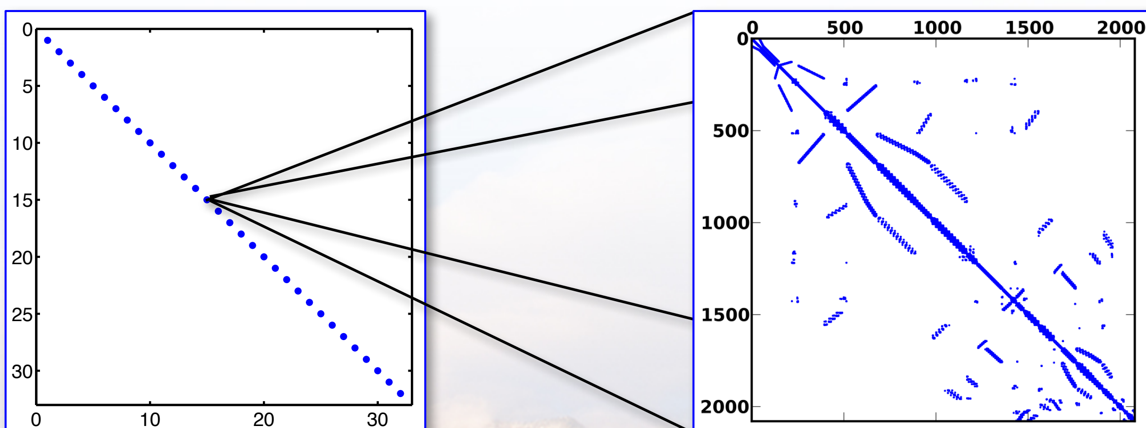
- PDE:

$$f(u, y) = 0$$

- Propagating *m* samples – block diagonal (nonlinear) system:

$$F(U,Y) = 0,\; U = \sum_{i=1}^{m} e_i \otimes u_i,\; Y = \sum_{i=1}^{m} e_i \otimes y_i,\; F = \sum_{i=1}^{m} e_i \otimes f(u_i, y_i),$$

$$\frac{\partial F}{\partial U} = \sum_{i=1}^{m} e_i e_i^T \otimes \frac{\partial f}{\partial u_i}$$




  – Spatial DOFs for each sample stored consecutively

# Ensemble Matrix-Vector Product

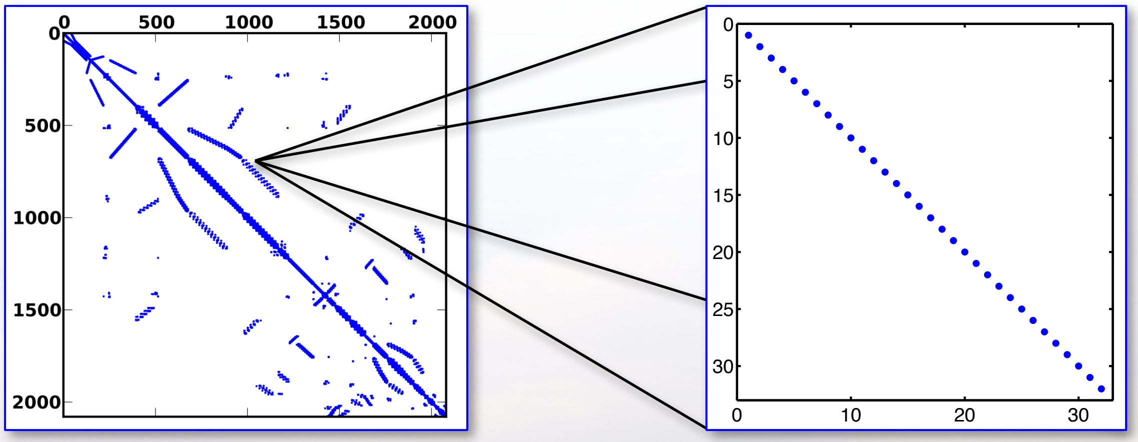
```
// Ensemble matrix-vector product
template <typename T, int m>
void ensemble_crs_mat_vec(const CrsMatrix<T>& A, const T *x, T *y) {
    for (int e=0; e < m; ++e) {
        for (int row=0; i<A.num_rows; ++row) {
            const int entry_begin = A.row_map[row];
            const int entry_end   = A.row_map[row+1];
            T sum = 0.0;
            for (int entry = entry_begin; entry < entry_end; ++entry) {
                const int col = A.col_entry[entry];
                sum += A.values[entry + e*A.num_entries] * x[col + e*A.num_rows];
            }
            y[row + e*A.num_rows] = sum;
        }
    }
}
```

# Simultaneous ensemble propagation

- Commute Kronecker products:

$$\tilde{F}(\tilde{U},\tilde{Y}) = 0,\ \tilde{U} = \sum_{i=1}^{m} u_i \otimes e_i,\ \tilde{Y} = \sum_{i=1}^{m} y_i \otimes e_i,\ \tilde{F} = \sum_{i=1}^{m} f(u_i, y_i) \otimes e_i,$$

$$\frac{\partial \tilde{F}}{\partial \tilde{U}} = \sum_{i=1}^{m} \frac{\partial f}{\partial u_i} \otimes e_i e_i^T$$




  - *m* sample values for each DOF stored consecutively

# Commuted, Ensemble Matrix-Vector Product

```
// Ensemble matrix-vector product using commuted layout
template <typename T, int m>
void ensemble_commuted_crs_mat_vec(const CrsMatrix<T>& A, const T *x, T *y) {
    for (int row=0; i<A.num_rows; ++row) {
        const int entry_begin = A.row_map[row];
        const int entry_end   = A.row_map[row+1];
        T sum[m];
        for (int e=0; e < m; ++e)
            sum[e] = 0.0;
        for (int entry = entry_begin; entry < entry_end; ++entry) {
            const int col = A.col_entry[entry];
            for (int e=0; e < m; ++e) {
                sum[e] += A.values[entry*m + e] * x[col*m + e];
            }
        }
        for (int e=0; e < m; ++e)
            y[row*m + e] = sum[e];
    }
}
```

- Automatically reuse non-sample dependent data
- Sparse access latency amortized across ensemble
- Math on ensemble naturally maps to vector arithmetic
- Communication latency amortized across ensemble

# C++ Ensemble Scalar Type

```
// Ensemble scalar type
template <typename U, int m>
struct Ensemble {
    U val[m];
    Ensemble(const U& v) { for (int e=0; e<m; ++e) val[m] = v; }
    Ensemble& operator=(const Ensemble& a) {
        for (int e=0; e<m; ++e) val[m] = a.val[m];
        return *this;
    }
    Ensemble& operator+=(const Ensemble& a) {
        for (int e=0; e<m; ++e) val[m] += a.val[m];
        return *this;
    }
    // ...
};

template <typename U, int m>
Ensemble<U,m> operator*(const Ensemble<U,m>& a, const Ensemble<U,m>& b) {
    Ensemble<U,m> c;
    for (int e=0; e<m; ++e) c.val[e] = a.val[e]*b.val[e];
    return c;
}

// ...
```

# Ensemble Matrix-Vector Product Through Operator Overloading

- Original matrix-vector product routine, instantiated with T = Ensemble<double,m> scalar type:

```
// Serial Crs matrix-vector product for arbitrary floating-point type T
template <typename T>
void crs_mat_vec(const CrsMatrix<T>& A, const T *x, T *y) {
    for (int row=0; row<A.num_rows; ++row) {
        const int entry_begin = A.row_map[row];
        const int entry_end   = A.row_map[row+1];
        T sum = 0.0;
        for (int entry = entry_begin; entry < entry_end; ++entry) {
            const int col = A.col_entry[entry];
            sum += A.values[entry] * x[col];
        }
        y[row] = sum;
    }
}
```

# Stokhos: Trilinos Tools for Embedded UQ Methods

- Provides ensemble scalar type
  - Uses expression templates to fuse loops

$$d = a \times b + c = \{a_1 \times b_1 + c_1, \ldots, a_m \times b_m + c_m\}$$

http://trilinos.sandia.gov

- Enabled in simulation codes through template-based generic programming
  - Template C++ code on scalar type
  - Instantiate template code on ensemble scalar type

- Integrated with Kokkos (Edwards, Sunderland, Trott) for many-core parallelism
  - Specializes Kokkos data-structures, execution policies to map vectorization parallelism across ensemble

- Integrated with Tpetra-based solvers for hybrid (MPI+X) parallel linear algebra
  - Exploits templating on scalar type
  - Krylov solvers (Belos)
  - Algebraic multigrid preconditioners (MueLu)
  - Incomplete factorization, polynomial, and relaxation-based preconditioners/smoothers (Ifpack2)
  - Sparse-direct solvers (Amesos2)

# Techniques Prototyped in FENL Mini-App*

http://trilinos.sandia.gov

- Simple nonlinear diffusion equation

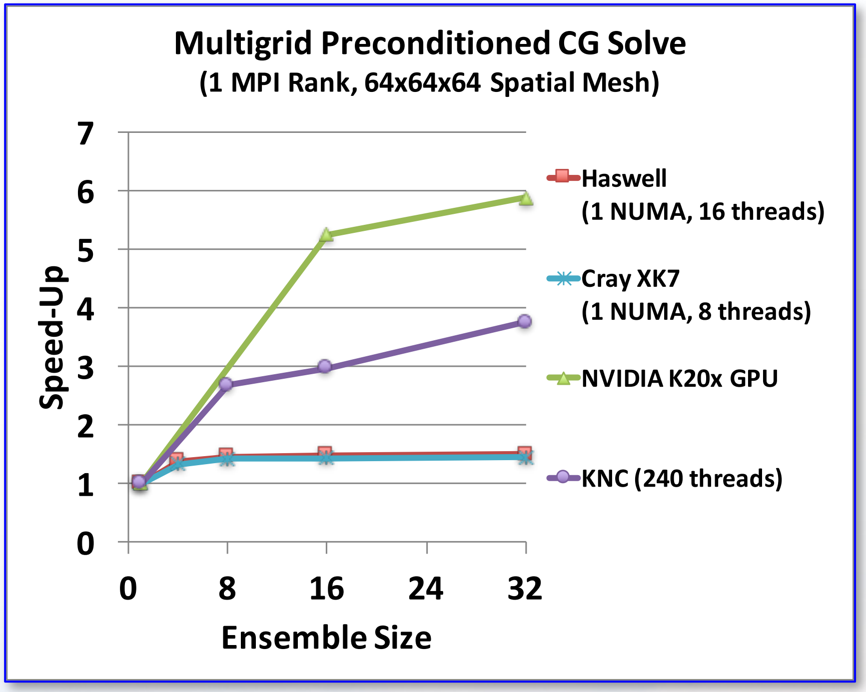
$$-\nabla \cdot (\kappa(x,y)\nabla u) + u^2 = 0,$$

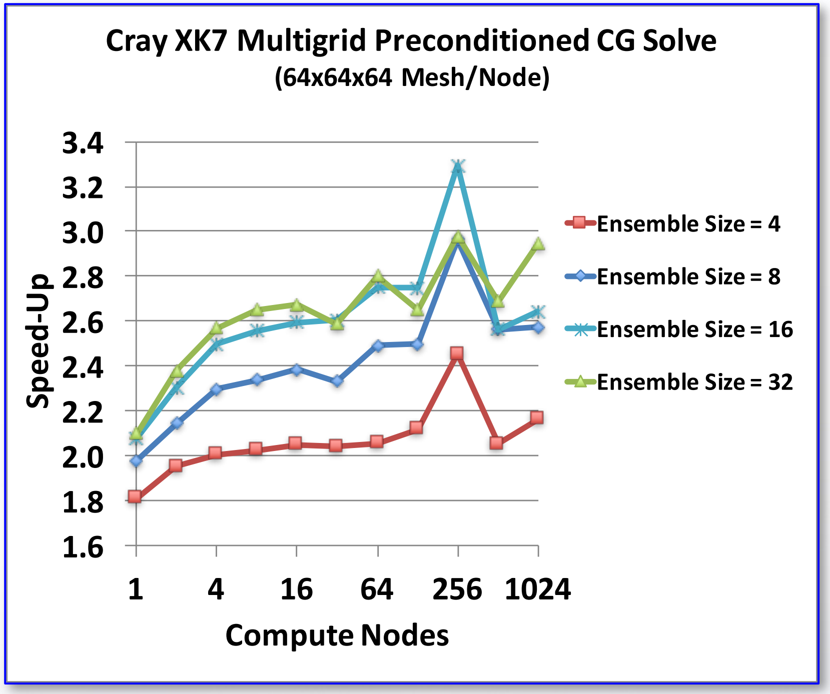
$$\kappa(x,y) = \kappa_0 + \sigma \sum_{i=1}^{M} \sqrt{\lambda_i}\kappa_i(x)y_i$$

  - 3-D, linear FEM discretization
  - 1x1x1 cube, unstructured mesh
  - KL truncation of exponential random field model for diffusion coefficient
  - Trilinos-couplings package
- Hybrid MPI+X parallelism
  - Traditional MPI domain decomposition using threads within each domain
- Employs Kokkos for thread-scalable
  - Graph construction
  - PDE matrix/RHS assembly
- Employs Tpetra for distributed linear algebra
  - CG iterative solver (Belos package)
  - Smoothed Aggregation AMG preconditioning (MueLu)
- Supports embedded ensemble propagation via Stokhos through entire assembly and solve
  - Samples generated via local and global sparse grids (TASMANIAN)

*Phipps, et al, *Embedded Ensemble Propagation for Improving Performance, Portability and Scalability of Uncertainty Quantification on Emerging Computational Architectures*, SISC, 2017

# AMG Preconditioned CG Solve



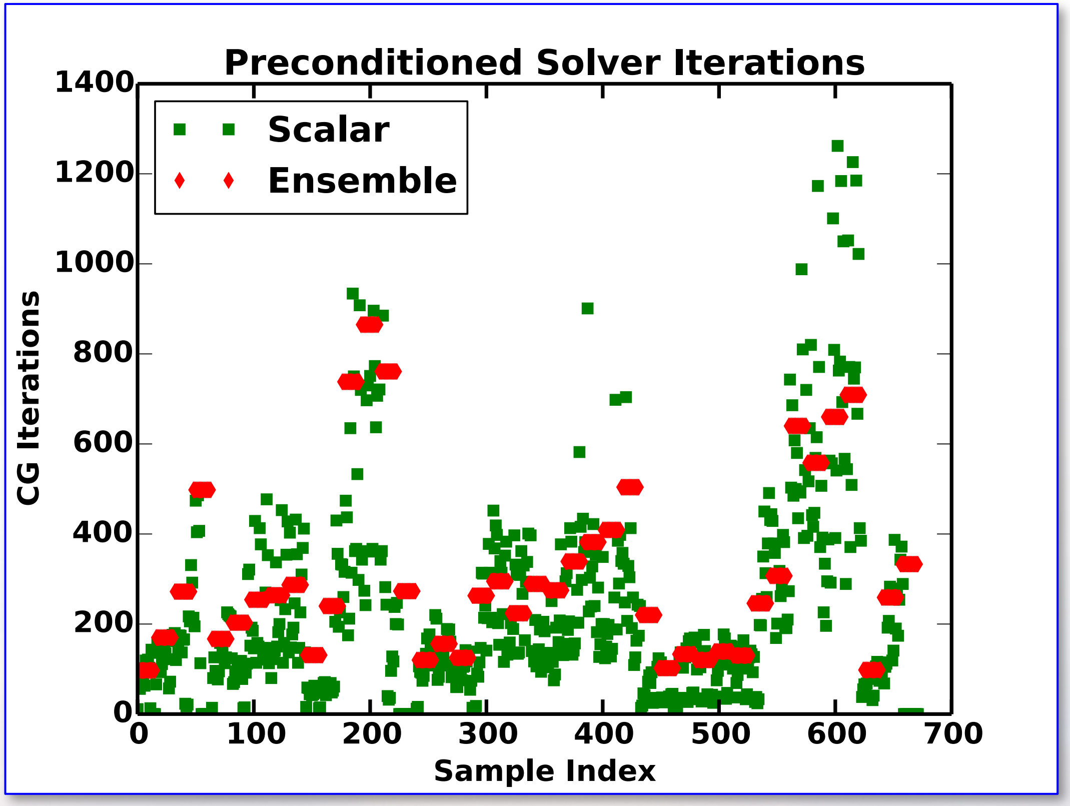

$$\text{Speed-Up} = \frac{\text{Ensemble size} \times \text{Time for single sample}}{\text{Time for ensemble}}$$

- Smoothed-aggregation algebraic multigrid preconditioning (MueLu)
  - Chebyshev smoothers
  - Sparse-direct coarse-grid solver (Amesos2/Basker)
  - Multi-jagged parallel repartioning (Zoltan2)

# Highly Anisotropic Diffusion

$$-\nabla \cdot (K(x,y)\nabla u) + u^2 = 0,$$
$$K(x,y) = \mathrm{diag}(\kappa(x,y), 1, 1)$$
$$\kappa(x,y) = 1 + 100\exp(\sqrt{300}\sum_{i=1}^{M}\sqrt{\lambda_i}\kappa_i(x)y_i))$$

- Decision on how to group samples will strongly impact performance



# Ensemble Grouping

- For these problems, computational work driven by the number of (preconditioned) solver iterations
- Special case of "ensemble divergence", where different samples in the ensemble would take diverging code paths
  - Biggest challenge for effective use of ensembles on hard problems
- Solution: group samples to minimize divergence
  - In this case, group samples requiring similar numbers of iterations
- Challenge: we don't know the number of iterations beforehand

# Solution 1: Expert Knowledge*

- Use expert knowledge on how the uncertain parameters affect linear solver convergence
  – For highly anisotropic diffusion equation, convergence is highly correlated with level of anisotropy:

$$a(y) = \max_{x} \left[ \frac{\lambda_{\max}(K(x, y))}{\lambda_{\min}(K(x, y))} \right]$$

- Grouping algorithm:
  – Compute anisotropy *a(y)* for each sample *y*
  – Sort samples based on increasing *a*
  – Divide sorted list in ensembles of given size *m*

- Note, evaluation of *a(y)* requires modification of the simulation code

*M. D'Elia, et al, *Ensemble Grouping Strategies for Embedded Stochastic Collocation Methods Applied to Anisotropic Diffusion Problems,* submitted to JUQ, 2016.

# Solution 2: Iterations Surrogate*

- For adaptive sampling methods, use previous samples to predict iterations for future samples
  - E.g., locally adaptive sparse grids
  - Use surrogate generated from previous samples evaluated on new samples
- Grouping algorithm (for adaptive sparse grids):
  - Build interpolant over previous sparse grid levels for linear solver iterations
  - Evaluate interpolant for samples at new level
  - Sort samples based on increasing iterations surrogate
  - Divide sorted list into ensembles of size *m*
- Note:
  - For first level, just use natural ordering of samples (no grouping)
  - Requires ability to track when a sample would have converged when not part of an ensemble (which can be done)

*M. D'Elia, et al, *Surrogate-based Ensemble Grouping Strategies for Embedded Stochastic Collocation Methods,* submitted to JUQ, 2017 (arXiv: 1705.02003).

# Numerical Tests

- FENL mini-app to test performance of grouping methods
  - Highly anisotropic diffusion tensor
  - Ensemble propagation using Trilinos infrastructure
  - AMG (MueLu), CG (Belos)

- Locally adaptive sparse grids provided by TASMANIAN (http://tasmanian.ornl.gov)

- Measure of increased computational work:

$$R = \frac{S \sum_{e=1}^{N_e} I_e}{\sum_{k=1}^{N} i_k}$$

$S$: ensemble size
$N$: number of samples
$N_e$: number of ensembles $\approx N/S$
$I_e$: number of iterations for $e$th ensemble
$i_k$: number of iterations for $k$th sample

  - *R = 1* if all samples in ensemble take same number of iterations
  - In general *R > 1*
  - Ensemble speedup inversely proportional to *R*

Sandia National Laboratories

# Continuous Test Case

$$-\nabla \cdot (K(x,y)\nabla u) + u^2 = 0,$$
$$x \in [0,1]^3, \quad y \in [-1,1]^4$$
$$K(x,y) = \mathrm{diag}(\kappa(x,y),1,1)$$
$$\kappa(x,y) = 1 + 100 \exp\left(\sqrt{300}\sum_{i=1}^{4}\sqrt{\lambda_i}\kappa_i(x)y_i\right)$$

| I | $S$ | $R_1$ | $R_2$ | $R_3$ | $R_4$ | $R_5$ | $R_6$ | $R_7$ | $R_8$ | $R$ | Speed-up | Pred. Speed-up |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| its | 4 | 1.68 | 1.43 | 1.23 | 1.05 | 1.01 | 1.04 | 1.06 | 1.10 | 1.06 | – | 2.56 |
| sur | 4 | 2.04 | 1.44 | 1.27 | 1.32 | 1.13 | 1.09 | 1.10 | 1.14 | 1.15 | 2.35 | 2.37 |
| par | 4 | 2.04 | 1.71 | 1.52 | 1.30 | 1.34 | 1.20 | 1.12 | 1.12 | 1.24 | 1.81 | 2.20 |
| nat | 4 | 2.04 | 1.66 | 1.44 | 1.23 | 1.34 | 1.28 | 1.25 | 1.24 | 1.29 | 2.15 | 2.11 |
| its | 8 | 2.83 | 1.60 | 1.27 | 1.08 | 1.10 | 1.08 | 1.10 | 1.44 | 1.14 | – | 3.44 |
| sur | 8 | 2.83 | 1.67 | 1.33 | 1.14 | 1.13 | 1.11 | 1.12 | 1.44 | 1.17 | 3.35 | 3.35 |
| par | 8 | 2.83 | 2.15 | 1.71 | 1.39 | 1.49 | 1.27 | 1.18 | 1.47 | 1.35 | 2.84 | 2.89 |
| nat | 8 | 2.83 | 2.29 | 1.90 | 1.48 | 1.49 | 1.51 | 1.41 | 1.64 | 1.52 | 2.61 | 2.58 |
| its | 16 | 3.11 | 1.94 | 1.60 | 1.12 | 1.28 | 1.25 | 1.25 | 1.56 | 1.30 | – | 3.94 |
| sur | 16 | 3.11 | 1.94 | 1.69 | 1.19 | 1.30 | 1.29 | 1.28 | 1.56 | 1.33 | 3.70 | 3.84 |
| par | 16 | 3.11 | 2.59 | 1.83 | 1.46 | 1.65 | 1.41 | 1.30 | 1.57 | 1.49 | 3.33 | 3.43 |
| nat | 16 | 3.11 | 2.59 | 2.12 | 1.87 | 1.80 | 1.83 | 1.67 | 2.16 | 1.84 | 2.73 | 2.78 |
| its | 32 | 6.22 | 3.07 | 2.62 | 1.25 | 1.67 | 1.32 | 1.61 | 2.63 | 1.66 | – | 3.46 |
| sur | 32 | 6.22 | 3.07 | 2.75 | 1.30 | 1.70 | 1.36 | 1.64 | 2.64 | 1.70 | 3.47 | 3.39 |
| par | 32 | 6.22 | 3.07 | 2.76 | 1.59 | 2.11 | 1.57 | 1.65 | 2.64 | 1.87 | 3.05 | 3.08 |
| nat | 32 | 6.22 | 3.07 | 2.99 | 2.37 | 2.24 | 2.16 | 2.07 | 2.74 | 2.28 | 2.06 | 2.53 |

# Discontinuous Test Case

$$-\nabla \cdot (K(x,y)\nabla u) + u^2 = 0,$$
$$x \in [0,1]^3, \quad y \in [-1,1]^4$$
$$K(x,y) = \text{diag}(\kappa(x,y), 1, 1)$$
$$\kappa(x,y) = 1 + k(y) \exp\left(\sqrt{300}\sum_{i=1}^{4}\sqrt{\lambda_i}\kappa_i(x)y_i\right)$$

$$k(y) = \begin{cases} 1 & r(y) < \frac{d}{4} \\ 100 & \frac{d}{4} \le r(y) < \frac{d}{2} \\ 10 & r(y) \ge \frac{d}{2}, \end{cases}$$
$$r(y) = \|y\|_2, \quad d = \sqrt{3}$$

| I | $S$ | $R_1$ | $R_2$ | $R_3$ | $R_4$ | $R_5$ | $R$ | Speed-up | Pred. Speed-up |
|---|---|---|---|---|---|---|---|---|---|
| its | 4 | 1.77 | 1.06 | 1.20 | 1.06 | 1.06 | 1.08 | – | 2.51 |
| sur | 4 | 2.08 | 1.13 | 1.24 | 1.14 | 1.25 | 1.22 | 2.09 | 2.23 |
| par | 4 | 2.08 | 1.44 | 1.62 | 1.36 | 1.37 | 1.41 | 1.93 | 1.94 |
| nat | 4 | 2.08 | 1.51 | 1.32 | 1.34 | 1.35 | 1.36 | 2.00 | 2.01 |
| its | 8 | 2.91 | 1.23 | 1.56 | 1.16 | 1.14 | 1.22 | – | 3.22 |
| sur | 8 | 2.91 | 1.29 | 1.64 | 1.27 | 1.21 | 1.30 | 2.80 | 3.02 |
| par | 8 | 2.91 | 1.74 | 2.01 | 1.49 | 1.79 | 1.74 | 2.29 | 2.25 |
| nat | 8 | 2.91 | 1.56 | 1.80 | 1.55 | 1.55 | 1.59 | 2.41 | 2.46 |
| its | 16 | 3.33 | 1.79 | 1.64 | 1.22 | 1.17 | 1.29 | – | 3.97 |
| sur | 16 | 3.33 | 1.79 | 1.69 | 1.33 | 1.24 | 1.36 | 3.22 | 3.74 |
| par | 16 | 3.33 | 2.38 | 2.37 | 1.60 | 1.66 | 1.77 | 2.87 | 2.88 |
| nat | 16 | 3.33 | 2.38 | 2.10 | 1.99 | 1.81 | 1.93 | 2.60 | 2.65 |
| its | 32 | 6.65 | 2.88 | 2.28 | 1.38 | 1.28 | 1.54 | – | 3.74 |
| sur | 32 | 6.65 | 2.88 | 2.34 | 1.46 | 1.37 | 1.62 | 3.04 | 3.55 |
| par | 32 | 6.65 | 2.88 | 2.53 | 1.75 | 1.77 | 1.94 | 2.87 | 2.96 |
| nat | 32 | 6.65 | 2.88 | 2.87 | 2.56 | 2.16 | 2.43 | 2.39 | 2.38 |

# Summary

- Embedded sampling approach improves aggregate UQ performance by
  - Eliminating sparse memory accesses
  - Amortizing communication/access latency
  - Perfect fine-grained vector/Cuda-thread parallelism
- Applying technique through C++ templates greatly facilitates implementation
  - Alleviate code developers from having to worry about UQ
- Smart grouping of samples into ensembles required for more challenging problems
  - "Surrogate" approach works well for adaptive UQ methods, easily generalizable
- Working on new approach that adaptively choses samples to improve simulation QoI and minimize same divergence
  - Voronoi Piecewise Surrogates method of Ebedia and Rushdi

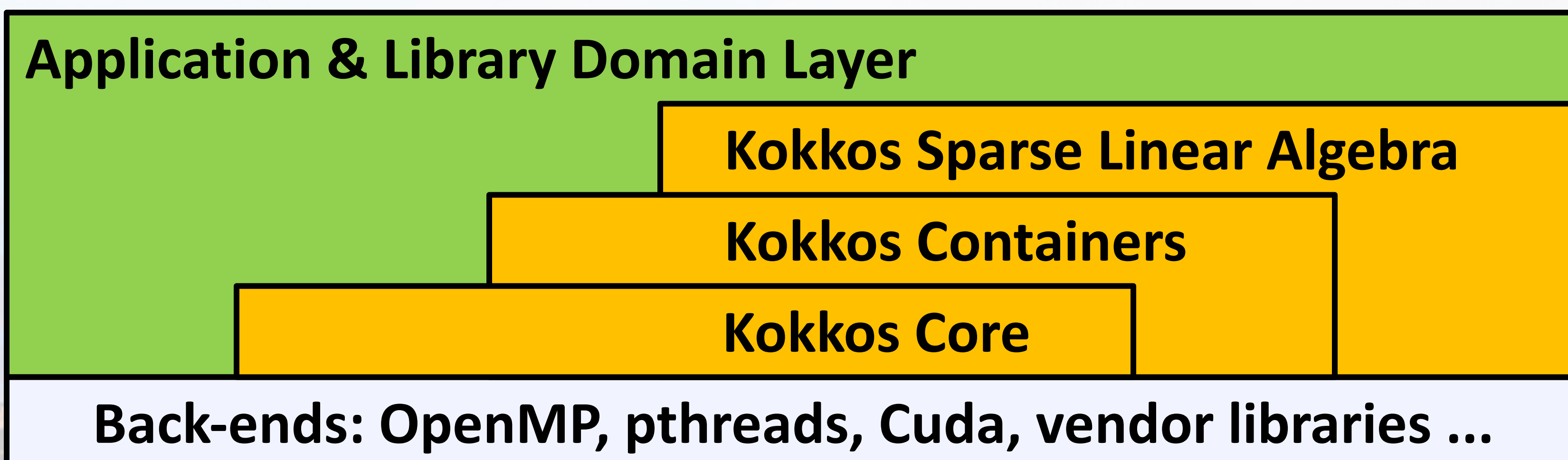
# Extra Slides

# Kokkos: A Manycore Device Performance Portability Library for C++ HPC Applications*

- Standard C++ library, not a language extension
  - Core: multidimensional arrays, parallel execution, atomic operations
  - Containers: Thread-scalable implementations of common data structures (vector, map, CRS graph, …)
  - LinAlg: Sparse matrix/vector linear algebra
- Relies heavily on C++ template meta-programming to introduce abstraction without performance penalty
  - Execution spaces (CPU, GPU, …)
  - Memory spaces (Host memory, GPU memory, scratch-pad, texture cache, …)
  - Layout of multidimensional data in memory
  - Scalar type

http://trilinos.sandia.gov

*H.C. Edwards, D. Sunderland, C. Trott (SNL)



# Tpetra: Foundational Layer / Library for Sparse Linear Algebra Solvers on Next-Generation Architectures*

- Tpetra: Sandia's templated C++ library for distributed memory (MPI) sparse linear algebra
  - Builds distributed memory linear algebra on top of Kokkos library
  - Distributed memory vectors, multi-vectors, and sparse matrices
  - Data distribution maps and communication operations
  - Fundamental computations: axpy, dot, norm, matrix-vector multiply, ...
  - Templated on "scalar" type: float, double, automatic differentiation, polynomial chaos, ensembles, …

Trilinos

http://trilinos.sandia.gov

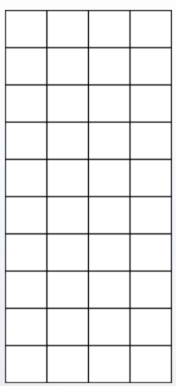
- Higher level solver libraries built on Tpetra
  - Preconditioned iterative algorithms (Belos)
  - Incomplete factorization preconditioners (Ifpack2, ShyLU)
  - Multigrid solvers (MueLu)
  - All templated on the scalar type

*M. Heroux, M. Hoemmen, *et al* (SNL)

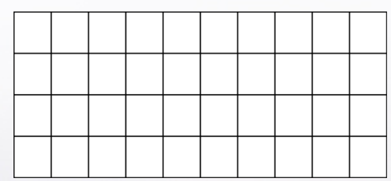
# Kokkos Integration

- Kokkos views of UQ scalar type internally stored as views of 1-higher rank
  - UQ dimension is always contiguous, regardless of layout
- Facilitates
  - Fine-grained parallelism over UQ dimension
  - Efficient allocation and initialization
  - Specialization of kernels
  - Transfering data between host and device and MPI communication

```
Kokkos::View< Ensemble<double,4>*, LayoutRight, Device > view(“v”, 10);
```

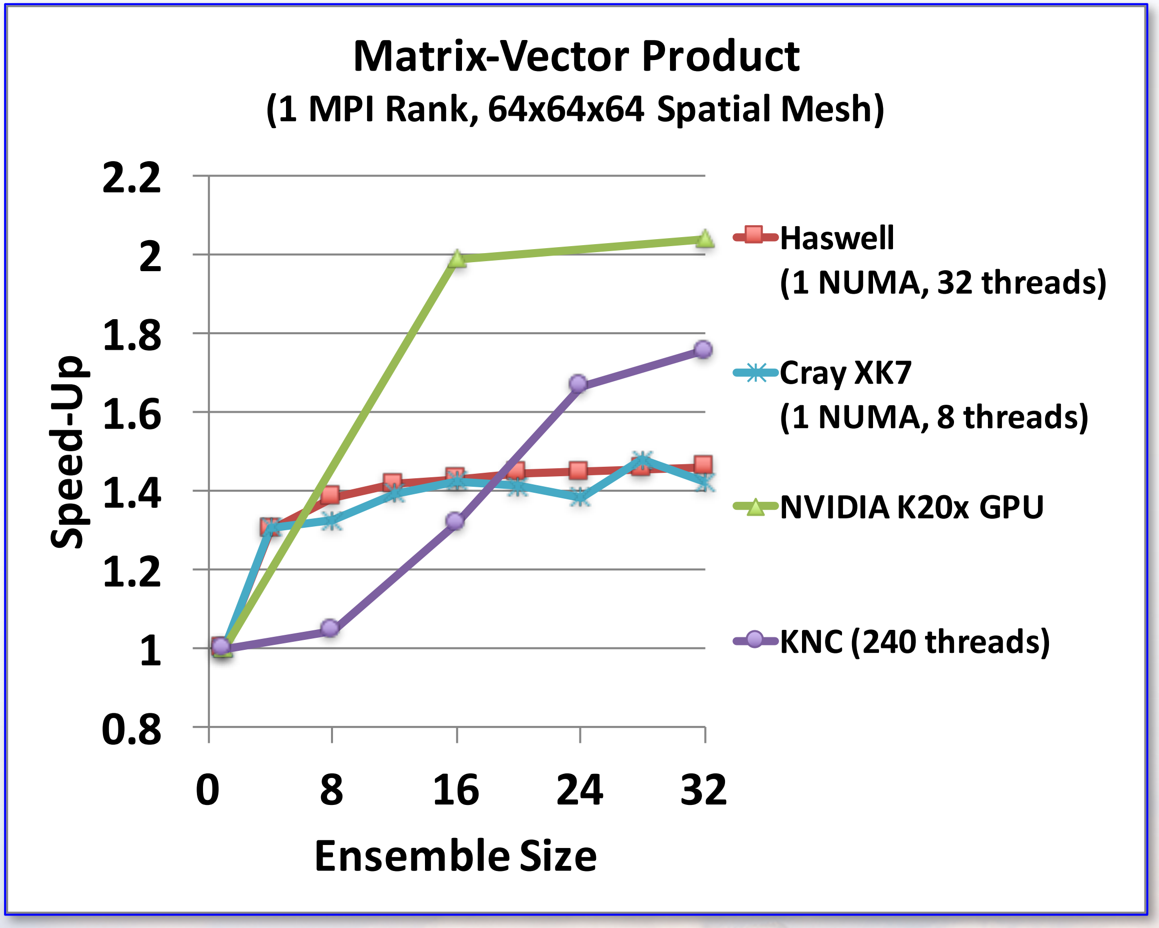


```
Kokkos::View< Ensemble<double,4>*, LayoutLeft, Device > view(“v”, 10);
```



- Requires specialized kernel launch for CUDA to map warp to UQ dimension to achieve performance

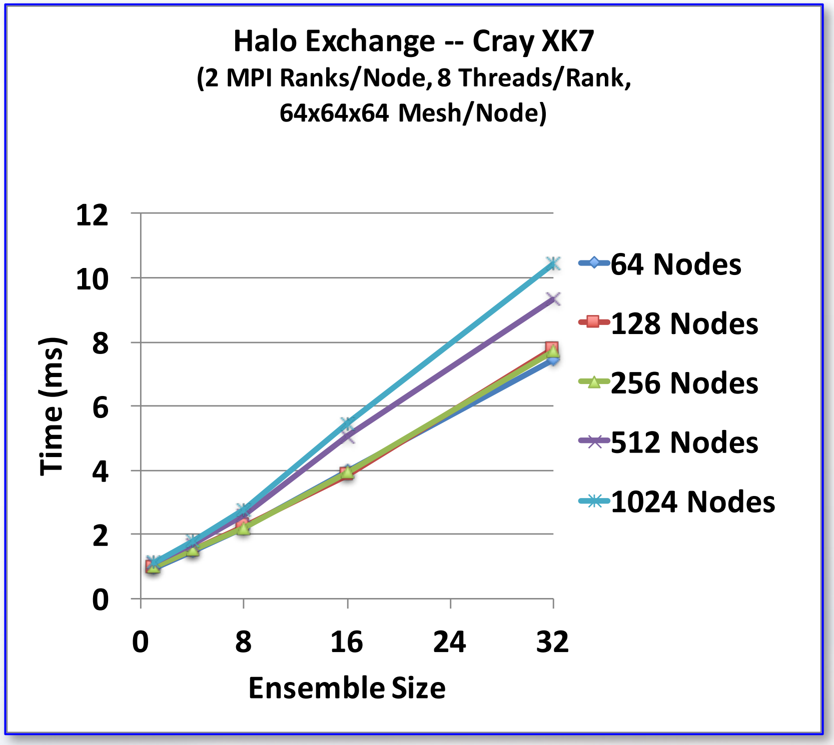
# Ensemble Sparse Matrix-Vector Product Speed-Up

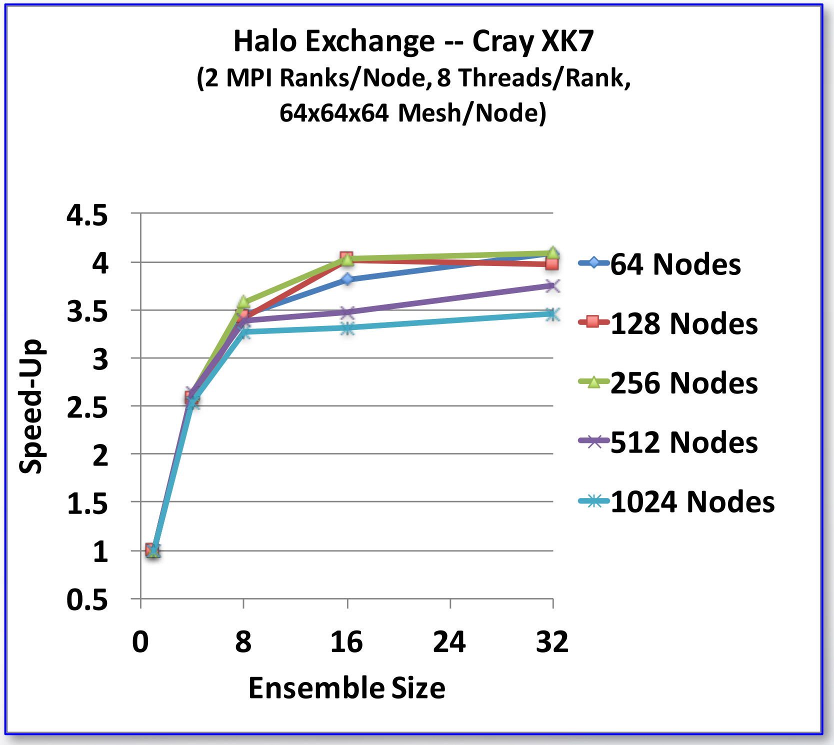



- Speed-up results from
  - Reuse of matrix graph
  - Replacement of sparse gather with contiguous load
  - Perfect vectorization of multiply-add

$$\text{Speed-Up} = \frac{\text{Ensemble size} \times \text{Time for single sample}}{\text{Time for ensemble}}$$

# Interprocessor Halo Exchange



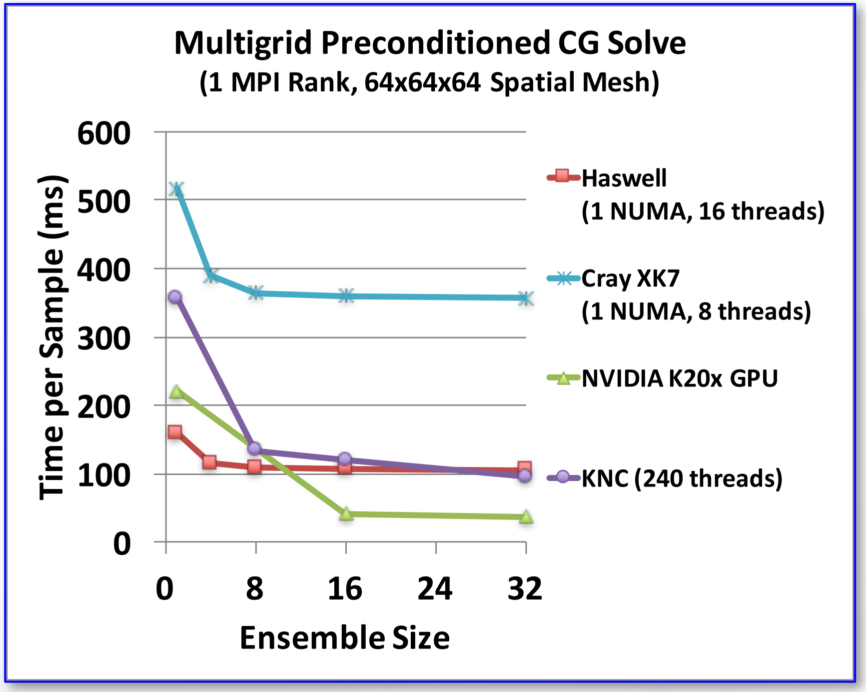

$$\text{Time} \approx a + bm$$

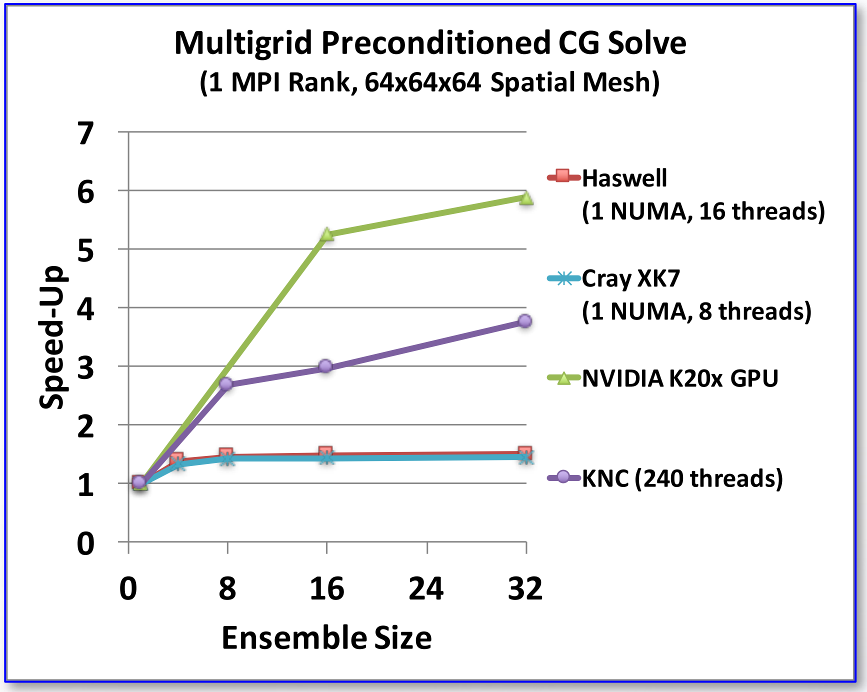
$$\text{Speed-Up} = \frac{\text{Ensemble size} \times \text{Time for single sample}}{\text{Time for ensemble}} \approx \frac{m(a+b)}{a+bm}$$

- Speed-up results from reduced aggregate communication latency
  - Fewer, larger MPI messages
  - Communication volume is the same

Sandia National Laboratories

# AMG Preconditioned CG Solve

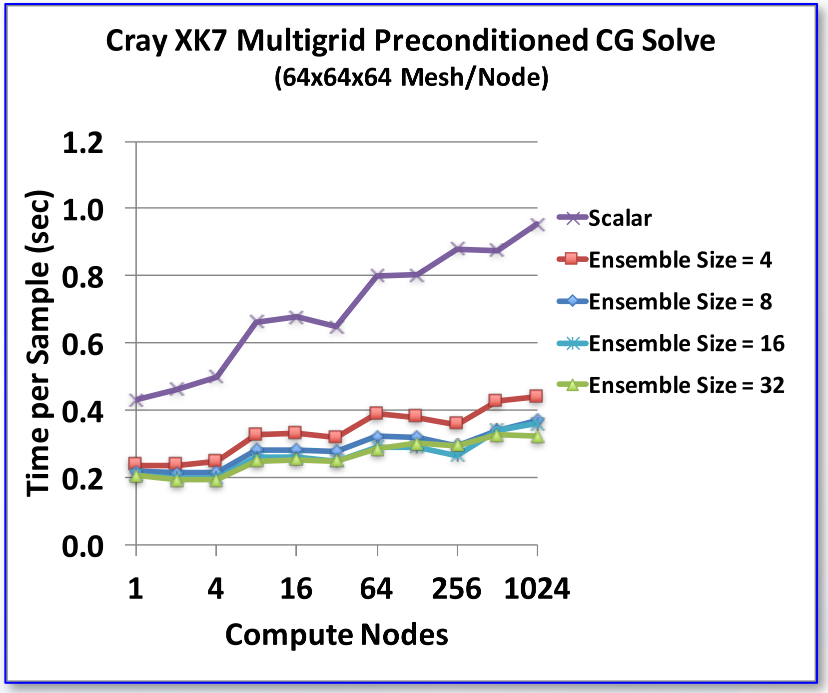


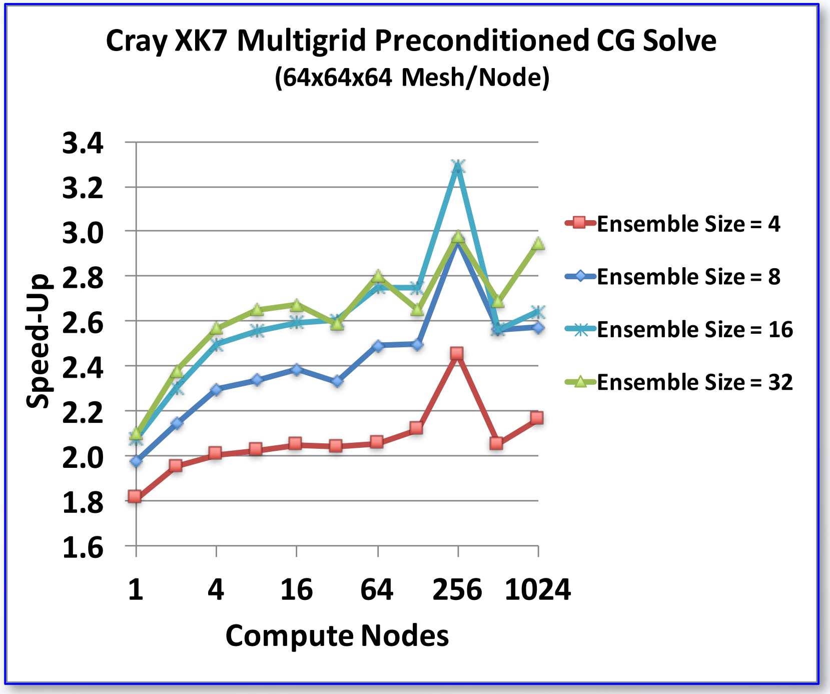




$$\text{Speed-Up} = \frac{\text{Ensemble size} \times \text{Time for single sample}}{\text{Time for ensemble}}$$

- Smoothed-aggregation algebraic multigrid preconditioning (MueLu)
  - Chebyshev smoothers
  - Sparse-direct coarse-grid solver (Amesos2/Basker)
  - Multi-jagged parallel repartioning (Zoltan2)

# AMG Preconditioned CG Solve

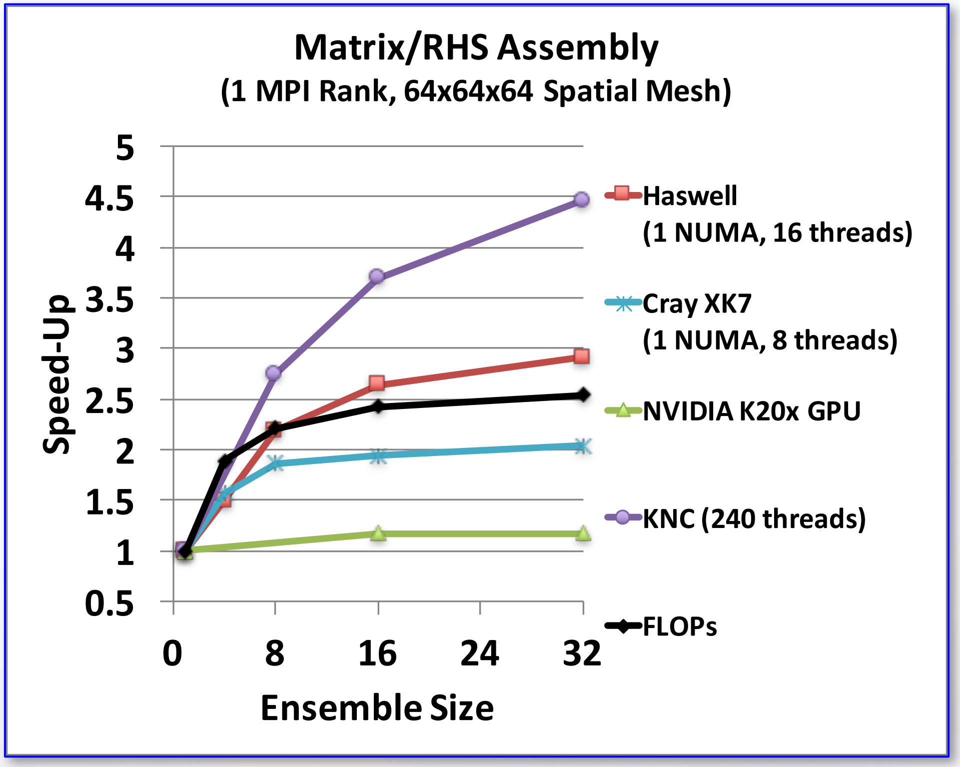



$$\text{Speed-Up} = \frac{\text{Ensemble size} \times \text{Time for single sample}}{\text{Time for ensemble}}$$

- Smoothed-aggregation algebraic multigrid preconditioning (MueLu)
  - Chebyshev smoothers
  - Sparse-direct coarse-grid solver (Amesos2/Basker)
  - Multi-jagged parallel repartioning (Zoltan2)

# Ensemble PDE Matrix/RHS Assembly Speed-Up




- Speed-up results from
  - Reuse of mesh, discretization data structures
  - Replacement of sparse gather with contiguous load
  - Perfect vectorization of math

$$\text{Speed-Up} = \frac{\text{Ensemble size} \times \text{Time for single sample}}{\text{Time for ensemble}}$$

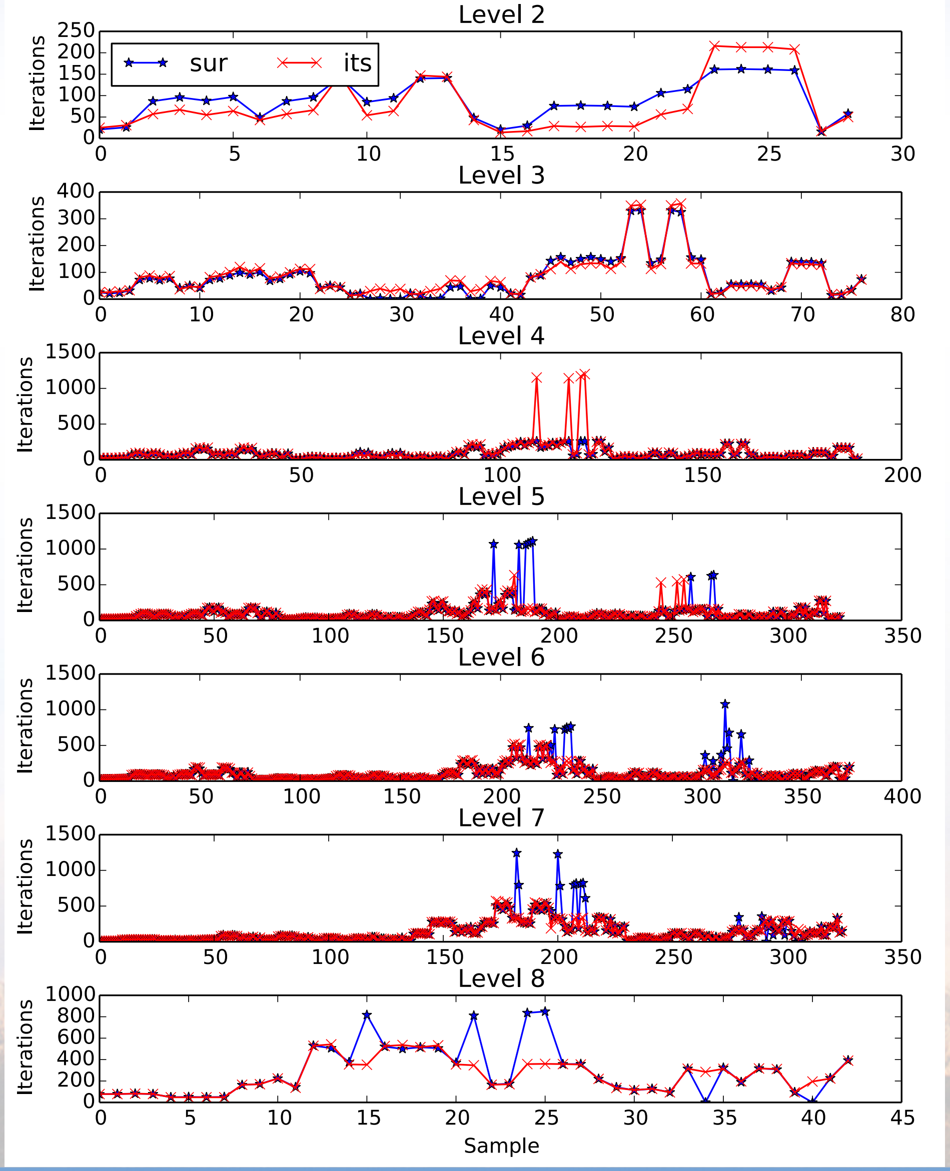
Sandia National Laboratories

# Ensemble Propagation for More Challenging Problems

- Assuming number of CG iterations doesn't vary significantly from sample to sample
  - True for problems with tame diffusion coefficient on regular meshes
  - Implies number of CG iterations for ensemble does not increase

- This is not true for many problems

# Continuous Test Case



# Discontinuous Test Case